



Android Security Analysis Final Report

Sponsor: NSA

Dept. No.: J83H

Contract No.: W56KGU-14-C-0010

Project No.: 0715N6CZ-AA

The views, opinions and/or findings contained in this report are those of the MITRE Corporation and should not be construed as an official government position, policy, or decision, unless designated by other documentation.

Approved for public release; distribution unlimited. 16-0202

©2016 The MITRE Corporation.
All rights reserved.

**Michael Peck
Gananand Kini
Andrew Pyles**

March 2016

Approved By

Christine Alicea, Project Leader

Date

Table of Contents

1	Introduction.....	4
2	Addressing Android App Vulnerabilities with the Android Lint Checker	5
2.1	App Data-in-Transit Vulnerabilities due to Insecure Certificate Validation or Hostname Validation	5
2.2	App Data-at-Rest Vulnerabilities due to Insecure File Permissions	7
2.3	Broadcast Receiver Vulnerabilities	7
2.4	Best Practices for Native Code	8
2.5	Demonstrating Effectiveness of Lint Checks	8
2.5.1	MITRE Secure Code Review Practice	8
2.5.2	F-Droid.....	9
2.6	Applying Lint Checks without Source Code Access	9
3	Enhancing Android OS Platform Security	11
3.1	Data-in-Transit Vulnerabilities	11
3.1.1	Certificate Pinning Manifest Attribute.....	11
3.1.2	Manifest Attributes to Prevent Overriding X509TrustManager and HostnameVerifier	12
3.2	Data-at-Rest Vulnerabilities due to Insecure File Permissions	12
3.3	Mitigations for Platform Exploitation Techniques	13
3.3.1	Preventing Dynamic Code Execution	13
3.3.1.1	Android WebView	15
3.3.1.2	ART.....	16
3.3.1.3	Dynamic Bytecode Execution.....	17
3.3.2	Limiting Privileges of the System Userid	17
3.4	KeyChain Improvements	17
4	Conclusion and Potential Future Work.....	19
5	References.....	20
Appendix A	Using the New Lint Checks	25
Appendix B	Demonstration Application.....	27
Appendix C	Android Runtime (ART) Additional Discussion	28
C.1	ART Background.....	28
C.2	JIT Compilation in ART.....	28
C.2.1	Motivation	28
C.2.2	Security Risks	29
Appendix D	Memory Mapping Examples	30
D.1	Memory Mappings of Chrome App	30
D.2	Memory Mappings of Application Using WebView	31

List of Figures

Figure 1: Example Lint Output Reporting a Vulnerable HostnameVerifier Implementation	5
Figure 2: Spoofed SMS Intent Example	9
Figure 3: Example of a Malicious App Downloading and Executing Exploit Code after Installation.....	14
Figure 4: Example of Using the KeyChain to Select a Key	18
Figure 5: Memory Mappings of Chrome App	30
Figure 6: Memory Mappings of Audible App	31

1 Introduction

According to recent worldwide sales figures reported by Gartner [1], Android is the most popular operating system (OS) when considering all general-purpose computing platforms (smartphones, tablets, laptops, and PCs). Mobile OSes such as Android introduce new security architectures designed with the experience of past lessons learned from traditional computing platforms. Most notably, Android provides a sandbox for applications (hereinafter “apps”) which isolates app data and code execution from other apps [2]. Android places security controls on allowed interactions between apps, and between each app and underlying device resources. The Android security architecture is designed to provide protection from malicious app behaviors, and to increase resilience to prevent or minimize the impact of exploitation of security vulnerabilities.

By default, apps cannot access data stored by another app, and are restricted from interfering with the behavior of another app. Apps must request permission to access device capabilities such as the microphone, camera, or physical location services, such as Global Positioning System (GPS). Apps also must request permission to access sensitive information repositories such as contact lists. Apps are also limited in their ability to access other underlying device resources and services. Every app must include a manifest file (`AndroidManifest.xml`) that defines the app’s permissions and other important properties. The contents of the manifest file are read and enforced by the Android OS.

Nevertheless, opportunities for exploiting app security vulnerabilities exist. Several types of vulnerabilities are commonly found in Android apps. These vulnerabilities, when present, can be exploited by resident malicious apps or by network-based attackers. Additionally, malicious apps may attempt to exploit Android platform vulnerabilities as a means of bypassing Android’s sandbox protections. This report describes the results of our research efforts to mitigate these issues by:

- Developing static analysis checks that allow app developers, security analysts, and app store operators to identify and eliminate common Android app vulnerabilities.
- Enhancing the Android OS to prevent common vulnerabilities from being exploited, to prevent use of common malicious app attack patterns, and to improve the security services provided to apps.

Our static analysis checks have been merged into the Android Open Source Project’s lint tool. They are present in the Android Studio 2.0 beta release and in the current beta release of the Android Plugin for Gradle. Our contribution has been noted in the Android Security Acknowledgements web page [46].

Enterprises are investing significant resources in app vetting personnel, tools, and techniques to determine whether apps are safe to deploy on their devices. Stronger understanding of the sandbox protections provided by the Android OS, and improvements to those protections where appropriate, will enable enterprises to efficiently allocate and prioritize their limited vetting resources based on the security protections already provided by the underlying device platform.

2 Addressing Android App Vulnerabilities with the Android Lint Checker

The Android Open Source Project includes a free, open source Software Development Kit (SDK) typically used by software developers to create Android apps. The SDK contains a code scanning tool called lint [3]. Lint scans app code using defined rules and alerts the developer to potential issues (including, but not limited to, security issues). Lint is integrated into Android Studio and Eclipse, the primary graphical environments used by Android app developers. Lint is also integrated into command-line Android app development tools such as gradle and ant. Thus, placing checks for common app security vulnerabilities in lint enables developers to easily identify and correct security mistakes up-front in the app development lifecycle, minimizing security risks and costs. Lint can also be used by security assessors who have access to source code.

We developed several new lint checks, discussed below, for common app security vulnerabilities. Our checks were accepted by the Android Open Source Project and are included in the Android Studio 2.0 beta release and in the current beta release of the Android Plugin for Gradle used by the Android SDK when compiling apps from the command line. Appendix A provides historical information, no longer needed, describing how to compile the checks as custom rules and incorporate them into the lint tool as a plugin. Appendix B provides details of an app with deliberately introduced vulnerabilities that we used to demonstrate the lint checks.

Figure 1 shows an example of output from the lint tool reporting a vulnerable `HostnameVerifier` implementation (as described below in Section 2.1.1). In this example, the lint tool's HyperText Markup Language (HTML) output is shown. Lint can also output EXtensible Markup Language (XML) or plaintext.

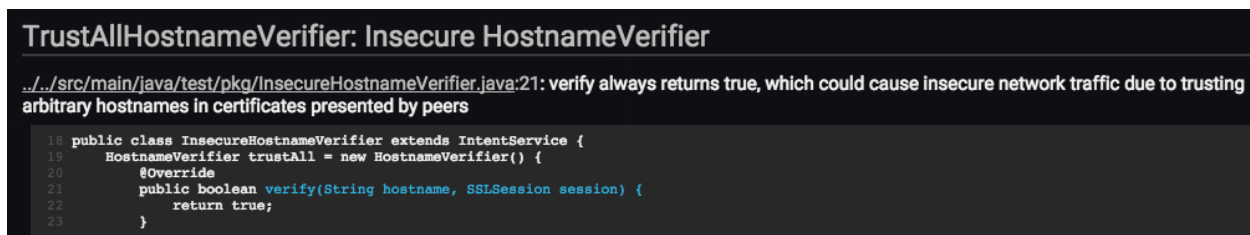


Figure 1: Example Lint Output Reporting a Vulnerable `HostnameVerifier` Implementation

2.1 App Data-in-Transit Vulnerabilities due to Insecure Certificate Validation or Hostname Validation

Numerous efforts have documented common failures by Android apps to properly check X.509 certificates while establishing Secure Socket Layer (SSL) and Transport Layer Security (TLS) sessions, making the network connections susceptible to man-in-the-middle (MITM) attacks:

- Fahl, et al. [4] statically analyzed 13,500 popular free apps from the Google Play Store and found that 1,074 apps contain SSL specific code that either accepts all certificates, or accepts all hostnames for a certificate, and thus are vulnerable to MITM attacks. They performed a manual audit of 100 apps and found that 41 of those 100 were vulnerable to MITM attacks.

- Sounthiraraj, et al. [5] statically analyzed 23,418 apps from the Google Play Store to similarly search for `TrustManager` and `HostnameVerifier` issues, finding 1,453 potentially vulnerable apps. They applied dynamic analysis to those apps and confirmed 726 to be vulnerable to MITM attacks.
- FireEye [6] reviewed “the 1,000 most-downloaded free apps in the Google Play Store as of July 17, 2014.” They found that of the 614 apps that use SSL/TLS to communicate with a remote server, 448 use `TrustManagers` that do not properly check certificates, and 50 use `HostnameVerifiers` that do not properly check hostnames.
- Montelibano and Dormann [7] dynamically analyzed 1,000,500 Android apps using the CERT Tapioca tool, finding TLS-related vulnerabilities in 23,667 of the apps.

Additionally, as described by Grace et al. [8], network communications are in some cases used by apps to dynamically download new code for the app to execute. Grace et al., found this behavior in advertising libraries embedded into mobile apps, meaning that the app developer may not even be aware of the functionality and its potential security impact. Successful MITM attacks could provide remote code execution ability, as demonstrated by Ryan Welton of NowSecure against a keyboard app running with system-level privileges on many Samsung devices [9].

Android’s standard TLS library uses implementations of the `X509TrustManager` Java class to perform certificate validation. By default, an OS-provided `X509TrustManager` class is used, but apps have the ability to define and use their own `X509TrustManager`.

For example, app developers commonly, and legitimately, provide their own `X509TrustManager` to implement certificate pinning. Certificate pinning is the practice of defining a restricted list of trusted certificate authorities (CAs) for the app’s network connections rather than trusting all of the CAs in the default Android trust store. While this can prevent MITM attacks due to malicious certificates issued by rogue or compromised CAs, the risk of implementation mistakes increases when developers provide their own `X509TrustManager` instead of using the platform default implementation.

Similarly, Android’s standard TLS library uses implementations of the `HostnameVerifier` Java class to ensure that the hostname asserted by the other endpoint’s X.509 certificate matches the expected value. By default, an OS-provided `HostnameVerifier` is used, but apps have the ability to define and use their own `HostnameVerifier`. Just as with `X509TrustManager`, the risk of implementation mistakes increases when developers provide their own `HostnameVerifier` implementation rather than use the platform default implementation.

To guard against the use of `X509TrustManager` and `HostnameVerifier` implementations that bypass the desired security checks, we wrote 4 lint checks targeting trust management functionality. We wrote a lint check that identifies insecure `X509TrustManager` implementations whose `checkServerTrusted` or `checkClientTrusted` methods do nothing, thus causing any presented certificate chain to be trusted [10]. Acting on the suggestion of a Google engineer who reviewed our lint checks, we additionally also wrote a lint check for use of `SSLCertificateSocketFactory.getInsecure()`, which returns an `SSLSocketFactory` with certificate checks disabled [11]. We wrote a lint check that identifies insecure `HostnameVerifier` implementations whose `verify` method always returns true, thus trusting any hostname and making the connection susceptible to MITM attacks [12].

We also wrote a lint check for use of

`SSLCertificateSocketFactory.createSocket()` with an `InetAddress` (an IP address) as the first parameter rather than a DNS name, disabling hostname verification [11].

2.2 App Data-at-Rest Vulnerabilities due to Insecure File Permissions

Each Android app has its own internal storage directory. Linux file permissions are supposed to prevent apps from reading or writing files in another app's internal storage directory. However, an app may inadvertently set its file permissions to world-readable or world-writable, allowing other apps to read or manipulate the app's private files.

For example, file permission vulnerabilities were identified in the Skype app for Android in 2011 [13]. Skype stored sensitive information including account information and contact list information as both world-readable and world-writable.

More recently, NowSecure identified apps that store their own executable code with world-writable permissions [14], allowing a malicious app to overwrite the executable code and achieve the ability to execute malicious code with the privileges of the vulnerable app.

The Android SDK deprecated the ability for apps to use the Android `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE` flags to set insecure file permissions, and strongly discourages this practice in the Android developer documentation. Lint checks already exist to detect use of these flags by app developers, but do not cover all of the applicable methods. We expanded the coverage of the `MODE_WORLD_READABLE` and `MODE_WORLD_WRITEABLE` lint checks [15], and additionally introduced checks for use of the `java.io.File.setReadable()` and `java.io.File.setWritable()` methods [16], which can also be used in Android to set file permissions.

2.3 Broadcast Receiver Vulnerabilities

Android provides `Intents` as a means of communication between applications. An `Intent` can be used to invoke an Android `Activity`, `Service`, or `BroadcastReceiver`. An `Intent` can be declared as either explicit or implicit. An explicit `Intent` declares the destination application and component within that application. An implicit `Intent` does not declare an exact destination, but rather declares an action string and other information. Potential destination applications declare `Intent` filters in their `AndroidManifest.xml` file for the types of `Intents` they would like to receive. The Android OS uses this information to determine which destination components to deliver an implicit `Intent` to.

Android apps can define `BroadcastReceiver` components to receive and act upon broadcast intent messages sent from the Android OS or from other installed Android apps. We wrote lint checks to identify two common `BroadcastReceiver` vulnerabilities [17].

The first check identifies broadcast receivers that declare an `Intent` filter for a protected-broadcast action string but fail to actually check the action string in received broadcast `Intents`. `Intents` containing protected-broadcast action strings can only be sent by Android OS components, not by third-party apps. However, if the receiver simply assumes that the received broadcast intent contains the protected-broadcast action string without actually checking, then as described by Chin et al. [18], a malicious third-party app can inject its own broadcast intent into the broadcast receiver. This issue likely originates due to app developers mistakenly believing that `intent-filters` declared in their app manifests act as a security

mechanism, when in reality `intent-filters` are only used by Android to resolve the destination of implicit `Intents`. `Intent` senders can use explicit `Intents` to attempt to deliver `Intents` to any destination regardless of `intent-filter`.

The second check identifies broadcast receivers that declare an `intent-filter` for the `SMS_DELIVER` or `SMS_RECEIVED` action string but fail to ensure that the sender holds the `BROADCAST_SMS` permission. In these cases, a malicious third-party app can potentially inject broadcasts into the vulnerable app that would then be treated as if they were Short Message Service (SMS) messages received by the device [19]. This issue was mitigated beginning in Android 6.0 by adding the action strings to the protected-broadcast action string list [20] (however, if the receiver does not check the sender's permission, then it must be sure to check the received intent's action string).

2.4 Best Practices for Native Code

As a best practice, apps should place their native shared libraries in the “lib” directory within the app package. The Android package manager extracts these files into an app library directory in `/data/app-lib` that apps themselves cannot write to, forcing code updates to be distributed as updated app packages rather than permitting apps to directly modify their own code. The ability for apps to update their own code is an attack vector that has been observed in malicious apps and also a source of app security vulnerabilities, for example as described by [14].

Apps can use either the `load` method or `loadLibrary` method (in either the `java.lang.Runtime` class or the `java.lang.System` class) to load native code. The `load` method takes in an absolute path, allowing apps to load native code from any location. The `loadLibrary` path simply takes the library name itself and will only load libraries from authorized locations (the platform library locations `/system/lib` and `/vendor/lib` and the app's library directory in `/data/app-lib`).

We proposed a lint check to identify calls to `java.lang.Runtime.load()` and `java.lang.System.load()` and recommend the use instead of `java.lang.Runtime.loadLibrary()` and `java.lang.System.loadLibrary()` [50].

We additionally proposed a lint check to identify the presence of native code in the `assets` and `res` app build directories and recommend that the native code instead be placed so that it gets bundled into the compiled app package's `lib` directory.

These best practices, if followed by app developers, could improve the feasibility of removing the ability for apps to execute code in their internal storage directories in a future Android release. Our recommendations for this are described in more detail in Section 3.3.1.

2.5 Demonstrating Effectiveness of Lint Checks

The following sections discuss the effectiveness of the Android lint checks that we created for this effort.

2.5.1 MITRE Secure Code Review Practice

MITRE's Secure Code Review Practice analyzes software code for security flaws at the request of MITRE project leaders. We applied the Android lint tool, including our new checks, to three

Android apps submitted for code review in 2015 and 2016. Lint enabled us to identify a number of security issues that we subsequently reported to the developers.

2.5.2 F-Droid

F-Droid [21] is a repository of open source Android apps. In April 2015, we ran the Android lint tool against 981 apps obtained from the repository¹. Our new lint checks at the time (we developed additional checks since) identified potential security issues in 127 of the 981 apps. The combination of our lint checks and the security checks already included in the Android lint tool identified potential security issues in 568 of the 981 apps.

As an example, we identified a `BroadcastReceiver` vulnerability in an app that displays received SMS messages. The app does not verify that the sender holds the `BROADCAST_SMS` permission. The app also does not check that the received broadcast intent actually contains the `android.provider.Telephony.SMS_RECEIVED` action string. As shown in Figure 2, we were able to demonstrate using a malicious app to inject a spoofed SMS into the vulnerable app, which caused the SMS to be displayed to the user.

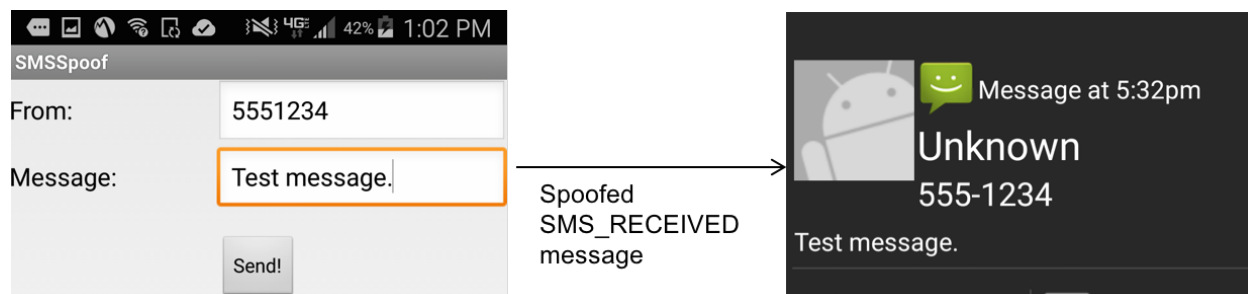


Figure 2: Spoofed SMS Intent Example

2.6 Applying Lint Checks without Source Code Access

We demonstrated a limited ability to apply the Android lint tool to compiled Android app packages (.apk files), rather than just on apps for which source code is available.

Android lint checks primarily analyze either Java source code files (.java files), compiled Java bytecode (.class files), app manifest files (`AndroidManifest.xml`) or a combination of the three². Without source code access, we could run the lint checks that operate on compiled Java bytecode or app manifest files and obtain output, but the checks that operate on Java source code files obviously do not produce any results since no Java source code files are available in this case.

In July 2015, we ran the Android lint tool (using the pre-existing security checks in the lint tool as well as the subset of our checks that were ready for use at the time) on 1726 Android app packages obtained from the Google Play Store. These were gathered from the top free 100 apps in each of the Play Store's 20 categories; some apps were in multiple categories or failed to download. We identified potential vulnerabilities in a large number of apps, including:

- Potential Transport Layer Security vulnerabilities (842 apps)

¹ F-Droid contained 1671 applications at the time of our analysis. We were only able to get 981 applications to successfully compile and provide lint output, so only those 981 applications were included in our analysis.

² Lint checks can also analyze application resource files as well as any arbitrary file within the application build directory.

- Does not declare `allowBackup` in the manifest. Defaults to true, perhaps unintentionally allowing the app's internal data to be backed up over USB using 'adb backup' (664 apps)
- Broadcast Receiver can potentially receive spoofed system broadcasts (417 apps)
- App was published with debug capabilities enabled (156 apps)
- Broadcast Receiver can potentially receive spoofed SMS broadcasts (36 apps)

Further detailed examination would be needed to determine if any of the potential vulnerabilities are actually exploitable.

Unfortunately, Android Studio's lint integration does not support the use of Java bytecode-based lint checks, which has led Google to move towards the use of Java source code-based lint checks and away from the use of Java bytecode-based lint checks. This practice will limit the effectiveness of applying the lint tool to compiled Android app packages.

Google takes steps of its own to assess the security of apps in the Google Play Store, likely by applying its own internal static analysis checks to compiled Android app packages. If issues are found, Google provides notifications to developers of the affected apps [59] [42] [43] [44].

3 Enhancing Android OS Platform Security

In the following section, we discuss changes that we proposed to the Android OS to make the platform more secure.

3.1 Data-in-Transit Vulnerabilities

A common security issue found in network communication of Android apps is communicating in cleartext rather than cryptographically protecting network communication using TLS or a similar protocol.

Android 6.0 introduced the `usesCleartextTraffic` manifest attribute³ [22] [23]. The attribute currently defaults to `true`, but when set to `false` declares the app developer's intention that the app should not perform any cleartext network communication. The Android OS will then make a best-effort attempt to prevent the app from using cleartext protocols such as HyperText Transfer Protocol (HTTP) rather than a cryptographically protected protocol such as HTTP Secure (HTTPS). The manifest attribute is enforced by Android's built-in network communication libraries. If apps themselves were to bundle in their own network communication implementations, the attribute would not necessarily be enforced.

Motivated by `usesCleartextTraffic`, we proposed several additional manifest attributes, detailed below. Google and the Android Open Source Project did not accept our proposals, responding that they already had an effort underway to provide similar capabilities. Their initial proposal was posted to their public code review system in early November 2015 [24] well after we had initiated our own investigations and has since been included in the Android N Developer Preview as the Network Security Configuration feature [57] [58].

3.1.1 Certificate Pinning Manifest Attribute

Inspired by previous work done by Tendulkar and Enck [25], we proposed [26] a `cert-pin` manifest attribute to enable app developers to declare certificate pins that should be used for TLS-protected network connections made by the app. As described above, app developers currently must write a custom `X509TrustManager` to implement certificate pinning. By allowing developers to specify certificate pins in the manifest and having those be automatically used by Android's built-in TLS implementation, developers would no longer need to write a custom `X509TrustManager`, eliminating a potential source of security bugs. Additionally, it is much simpler for security assessors to audit the manifest attribute contents than to audit the application source code or compiled bytecode.

Android currently includes certificate pinning support, but it (until the Android N Developer Preview) only supports a system-wide certificate pin list, can only be updated by an authorized entity such as Google, and does not allow per-app modifications to the list⁴. Our implementation built upon this already existing certificate pinning support but extended it to be customizable on a per-app basis.

³ Apple iOS 9, released around the same time as Android 6.0, added a similar new feature called App Transport Security that defaults to requiring all application network communication use HTTPS with TLS 1.2 and specific cipher suites (cryptographic algorithms). Unlike Android 6.0, Apple iOS 9's default behavior is to enable the feature.

⁴ Nikolay Elenkov provides a detailed description of Android's built-in certificate pinning functionality here: <http://nelenkov.blogspot.com/2012/12/certificate-pinning-in-android-42.html> (accessed 30 July 2015), with the caveat that his statement that "the standard `checkServerTrusted()` method doesn't consult the pin list" is no longer accurate. The `checkServerTrusted()` method in the default `X509TrustManager` does now use the pin list.

3.1.2 Manifest Attributes to Prevent Overriding X509TrustManager and HostnameVerifier

As previously described, custom `X509TrustManager` implementations are a common source of security vulnerabilities. We proposed an `allowTLSTrustManagerOverride` manifest attribute with an initial default value of `true`. When the variable is set to `false`, the app is declaring that it does not intend to override the Android platform's default TLS `TrustManager` implementation used to verify server certificates. The Android platform's TLS library will then ignore attempts by the app to override the `TrustManager`. This is a best-effort mechanism that would only be effective for apps that use Android's built-in TLS implementation, although third-party implementations could also choose to honor the flag. We implemented this attribute, modified Android's `conscrypt` library to make use of it, and demonstrated its effectiveness with a proof-of-concept app.

We additionally proposed a similar manifest attribute `allowTLSHostnameVerifierOverride` with an initial default value of `true` [28]. When `false`, the app is declaring that it does not intend to override the Android platform's default TLS `HostnameVerifier` implementation used to verify server hostnames. The Android platform's built-in library will then ignore attempts by the app to override the `HostnameVerifier`.

3.2 Data-at-Rest Vulnerabilities due to Insecure File Permissions

As previously described, Android apps may inadvertently set file permissions to world-readable or world-writable, potentially placing sensitive data at risk of being read or manipulated by other apps.

Security-Enhanced Linux (SELinux) mandatory access control policies can be used to prevent apps from reading/writing to the internal storage directory of other apps, regardless of file permissions, while still allowing Android's preferred methods of sharing data between apps such as `ContentProvider` to be used. However, adding SELinux policies introduces compatibility concerns for any apps that depend on the ability to set file permissions.

In an effort to improve app security while addressing the compatibility concerns, we proposed a new `isolatedAppData` app manifest attribute [29] to provide app developers with the ability to opt-in to have these stricter policies applied upon their apps. The attribute declares whether the platform should prevent read and write access to the app's internal data storage directory by other apps and also prevent the app from reading or writing to other apps' internal data storage directories regardless of assigned file permissions. The attribute's default value is `false`, but the default could be changed to `true` (or better yet, forced to `true`) in a future Android API level after providing app developers adequate warning and time to adjust their apps. We implemented this SELinux opt-in approach by appending a `“:isolated”` suffix to the app's `seinfo` string within the Android OS. The `seinfo` string is used by Android to determine what SELinux domain the app will be executed in and what domain will be used for the app's internal storage directory and files.

Unfortunately, our `isolatedAppData` app manifest attribute was not accepted. In January 2016, we proposed an alternate approach that, without introducing a new manifest attribute, could be used to automatically apply the policy based on the app manifest's `targetSdkVersion` field. We proposed a `minTargetSdkVersion` input selector be added for use in the `seapp_contexts` file [62]. The `seapp_contexts` file, part of the

device SELinux policy configuration, determines the SELinux domain and related policy configuration to be applied to each Android app based on the app's properties. The `minTargetSdkVersion` input selector could be used to phase in new SELinux policies for apps by giving app developers an opportunity to make any needed compatibility changes to their apps before updating each app's `targetSdkVersion` field in the app's `AndroidManifest.xml`. The `targetSdkVersion` field is used by the app developer to specify, for compatibility purposes, the highest version of Android (by API level) that the developer has tested the app with [60]. This general pattern has been used in the past for Android security improvements that could affect app compatibility. For example, Android Content Providers used to be exported (made available for use by other apps) by default, but this behavior was changed for apps with a `targetSdkVersion` of 17 or higher [61].

3.3 Mitigations for Platform Exploitation Techniques

In this section, we discuss techniques that a malicious user can employ to exploit the Android platform, and our recommended mitigations.

3.3.1 Preventing Dynamic Code Execution

A number of app vetting services exist that attempt to analyze Android app behavior for malicious activities, the most well-known being the Google Bouncer service used to assess apps that are submitted to the Google Play Store.

Malicious apps can evade app vetting by dynamically downloading and executing malicious code at execution time, for example as illustrated in Figure 3 below. Since the malicious code is not included in the app package that went through the vetting process, it likely will escape detection. This approach was suggested and demonstrated by Jon Oberheide [30], and described by Poeplau et al. [31]. A real-world example of this technique was found in materials allegedly leaked in July 2015 from Hacking Team, an Italian company that provides “easy-to-use offensive technology to the worldwide law enforcement and intelligence communities”. These leaked materials contained source code for an Android app that Hacking Team apparently placed in the Google Play Store, which downloads and executes platform exploit code after installation [32]. Maier et al. also describe their success using these techniques [33].

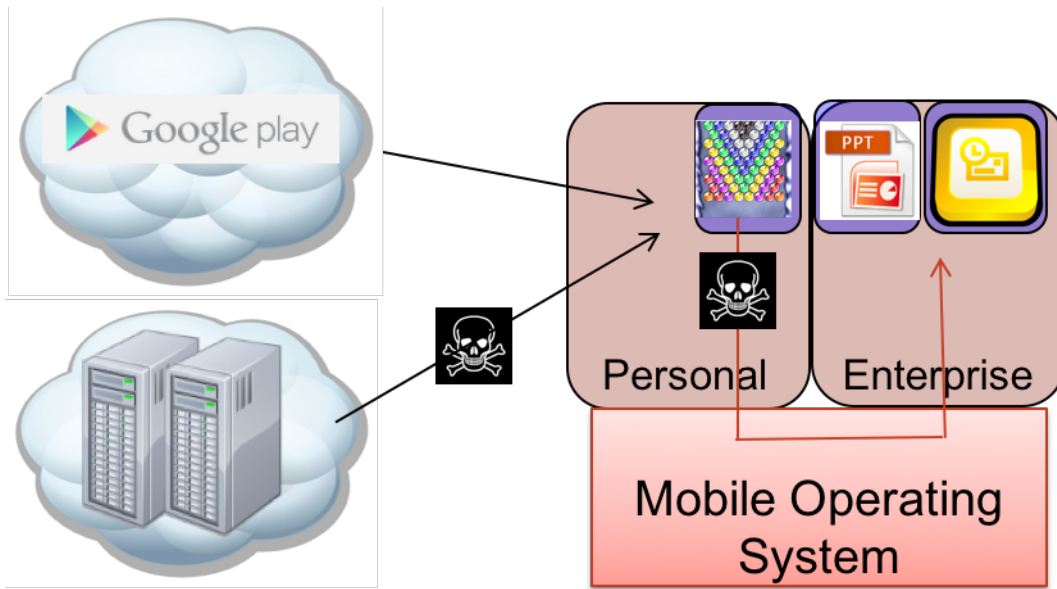


Figure 3: Example of a Malicious App Downloading and Executing Exploit Code after Installation

As a best practice, Android apps should place native shared libraries in the `lib` directory within the app package. At installation time, the Android package manager extracts these shared libraries into an app library directory in `/data/app-lib` that apps themselves cannot write to. This approach forces code updates to be distributed as app package updates, preventing apps from directly modifying their own code and enabling code review by app stores. However, nothing currently prevents Android apps at execution time from downloading native code, writing the code to a storage directory that the app has write permission to (such as the app's internal data storage directory), and then executing the code⁵.

SELinux mandatory access control policies can be added that block apps from having execute permission over any location that they can write to, forcing the developer to bundle executable code with the distributed app package where it is more feasible to inspect. In the master development branch of the Android Open Source Project, this policy was recently put in place for platform apps (apps signed by the vendor), but not yet for third-party apps, likely due to compatibility concerns. We believe the compatibility concerns are well-founded. In March 2014, we downloaded 2420 of the most popular free Android apps from the Google Play Store, chosen by selecting the top 100 free apps in each of the Play Store's 27 categories (some apps were listed in multiple categories and some apps failed to download). We found that 71 of the apps contained at least one native executable or shared library bundled in the app package in a directory other than `lib`, meaning they would likely be incompatible with this proposed policy. Other apps would be incompatible as well (but harder to detect with solely static checks) if they download or extract and then run executable code at run-time.

We proposed introducing an opt-in Android app manifest attribute `preventDownloadExecution` with a default value of `false` [34]. When set to `true`, the Android app is declaring to the platform that it does not intend to execute app-writable files, and

⁵ Similarly, applications could also place native code in other locations within their application package, such as the resource or assets directory, in an obfuscated or encrypted form, and then extract and execute the code at run time.

the platform would execute the app in a SELinux context that prohibits execution of app-writable files. The manifest attribute could be used as part of app inspection (for instance, by Google Bouncer or by an enterprise vetting system) to determine the potential risk level of the app and the rigor of evaluation that should be applied. The manifest attribute could also be phased in to potentially be true by default (or better yet, forced to always true) in a future Android release after giving app developers time to adjust their practices. Unfortunately, just as with our proposed `isolatedAppData` app manifest attribute discussed previously in section 3.2, our proposal was not accepted. We submitted a new proposal in January 2016 to make use of our `minTargetSdkVersion` proposal described in section 3.2 to phase in the stricter SELinux policy based on the `targetSdkVersion` declared in each app's `AndroidManifest.xml` [63] instead of defining a new manifest attribute.

Copperhead Security, a company developing an open source hardened Android distribution, has noted that this proposal, along with our proposal described in Section 3.2, “would be major game changers for the app security model” [51].

Even if apps are prevented from having execute privilege over app-writable files, apps still have the ability to map memory as both writable and executable, another vector allowing them to execute dynamic code. Copperhead Security proposes addressing this gap by adding W^X memory protections (requirements that the same memory regions cannot be both writable and executable) to Android using the PaX MPROTECT feature, roughly equivalent to removing SELinux `execmem` permission.

Unfortunately, enforcing W^X memory protections upon apps (whether using PaX MPROTECT, removing SELinux `execmem` permission, or another approach) introduces compatibility issues due to the use of just-in-time (JIT) compilation in the Android OS and potentially by apps themselves.

Android's Dalvik runtime, used until Android 4.4, regularly performs JIT compilation during app execution of each app's architecture-independent Dalvik code into optimized native code. Starting with Android 4.4, Android switched from Dalvik to the Android Runtime (ART). ART implements ahead-of-time (AOT) compilation, compiling portions of the app bytecode into optimized native code at app install time instead of at app run time, partly alleviating the need to allow apps to map memory regions as both writable and executable. Unfortunately, two main challenges continue to exist that prevent broad removal of `execmem` privilege from apps:

- Android's WebView capability, commonly used by apps, includes its own JIT compiler for optimizing web page JavaScript execution.
- ART still includes a JIT compiler that could potentially be invoked under some conditions.

Additionally, apps may include their own JIT compilation implementations.

3.3.1.1 Android WebView

Android WebView provides the capability to embed basic web browsing capabilities into Android apps. Android WebView is built from Chromium, Google's open source web browser, and includes a JIT compiler to optimize web page JavaScript execution. As shown in Appendix D.2, any app that uses WebView with JavaScript support enabled needs to have the ability to map memory as both writable and executable, making such an app incompatible with stronger executable memory protections.

Copperhead Security addressed this compatibility issue in its open source hardened Android distribution by automatically scanning apps for calls to `setJavaScriptEnabled()` and, if found, exempting the app from PaX MPROTECT [35]. However, we do not think this approach is practical for mainstream apps. We surveyed 1726 of the top free Google Play Store apps (the top 100 apps in each of the Play Store’s 20 categories, only counting apps once that are in more than one category, and minus apps that failed to download). We observed references to `android.webkit.WebView` in 1572 out of the 1726 apps. JavaScript is not enabled in `WebView` by default, but 1518 of the apps call `android.webkit.WebSettings.setJavaScriptEnabled()`. Many of these calls were found in app development frameworks such as Adobe AIR that are bundled into the app, making it unclear whether each app’s running functionality actually uses `WebView` (or JavaScript within the `WebView`). Regardless, Copperhead Security’s current approach would exempt almost all popular apps from its executable memory protections, severely decreasing the value of those protections.

`WebView` (and its JIT compiler) run with the privileges of the calling app. `WebView` uses a single process model to perform all actions including web page rendering and JavaScript interpretation [36]. The Chrome app for Android, which shares the same underlying Chromium code since Android 4.4, takes a different approach. The Chrome app uses Android’s `isolatedProcess` feature to run its JIT compiler (and other web page rendering processing) in a separate process with its own SELinux security context (`isolated_app`). In the case of the Chrome app, it appears to be feasible to impose executable memory protections upon the main app, allowing an exception only for the isolated rendering process. As shown in Appendix D.1, only the Chrome processes running in the `isolated_app` security context appear to map memory pages as both writable and executable. Since the isolated rendering process runs with very limited privileges, the potential harmful impact of an attacker taking advantage of its lack of executable memory protections would be limited.

We recommend exploring the feasibility of adopting Chrome’s isolated process model for Android `WebView`, so that JIT compilation is isolated into its own process running with a very limited security context. This approach would require adopting the multi-process model used by the Chrome Android app into `WebView`.

Alternatively, the performance impact of removing JIT compilation from `WebView` could also be explored. If `WebView`’s JavaScript performance is not severely impacted by removal of JIT compilation capabilities, then we would recommend removing it in order to improve the feasibility of enforcing stricter executable memory protections.

Until the current implementation of JIT within `WebView` is addressed, we do not believe it is practical to impose executable memory protections onto arbitrary Android apps.

3.3.1.2 ART

As described above, ART primarily performs ahead-of-time compilation of app bytecode into native code at app installation time and OS upgrade time, partly alleviating the need for apps to map memory regions as both writable and executable. However, for performance reasons, ART does not always actually compile all code ahead-of-time. Optimization options passed into the `dex2oat` command allow tradeoffs between compiling all, some, or none of the code ahead-of-time. By default, ART uses an interpreter to execute bytecode that has not been compiled ahead-of-time. Interpreters do not present a conflict with executable memory protections. However, ART still includes an optional JIT compilation capability, which does conflict with executable

memory protections. It appears that JIT compilation in ART is currently only enabled by default in Android engineering builds (special debug builds of the OS), so does not present a current concern, but would present a concern if ART's default behavior is changed in the future. Additional details are provided in Appendix C.

The Android N Developer Preview documentation describes changes in this area [64]. We have not examined the changes in detail or analyzed the impact.

3.3.1.3 Dynamic Bytecode Execution

Even if dynamic execution of native code from outside the system or app's library directories were to be blocked using SELinux policies, apps could still dynamically download and execute Dalvik bytecode through `DexClassLoader`. Apps could also embed their own interpreters directly into the app. The potential impact is unclear. Privilege escalation exploits appear to generally require native code execution capability. Further work is needed to assess the potential threats posed by bytecode.

3.3.2 Limiting Privileges of the System Userid

Many access control checks throughout the Android OS solely check that the caller holds the system (1000) Linux userid (uid). The system uid is used by numerous Android services and privileged platform apps. Currently, gaining malicious code execution as the system uid is disastrous to the overall security of the device, despite the movement of many of these system components into their own SELinux domains.

Changing these access control checks to instead use SELinux provides the ability to institute finer-grained checks. It also provides improved visibility into and the ability to centrally audit security policies.

As an example, the Android KeyChain service checks that the caller holds the system uid before allowing it to perform certain privileged operations. In reality, most of these privileged operations only need to be callable by the `DevicePolicyManagerService` (runs within the `system_server` domain), with a few operations that additionally may be called by the `com.android.settings` and `com.android.certinstaller` apps. We submitted a proposal to the Android Open Source Project to perform SELinux access control checks for these operations, along with SELinux policies granting the appropriate accesses to `system_server`, `com.android.settings`, and `com.android.certinstaller` [45]. Our proposal is still under consideration. This model could be applied to strengthen access control checks across the Android OS.

3.4 KeyChain Improvements

The Android OS provides a KeyChain service [37] [38] used to securely store private keys and make them available for use by installed apps. When available, devices can use hardware-backed security capabilities such as Trusted Execution Environments (TEEs) or Trusted Platform Modules (TPMs) to protect the private keys from disclosure, even if the Android OS is compromised. Figure 4 provides an example screenshot of the strongSwan IPsec Virtual Private Network (VPN) Client app requesting access to use a key stored in the KeyChain.

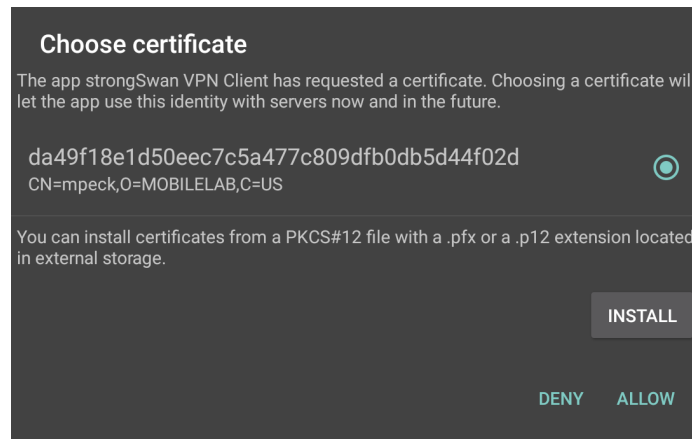


Figure 4: Example of Using the KeyChain to Select a Key

Unfortunately, Android currently does not provide API calls to generate keys within the KeyChain, forcing the keys to be generated elsewhere and imported in, unnecessarily increasing the risk of exposure of private keys.

We developed new calls to Android’s `DevicePolicyManager`, the API used by mobile device management systems, to enable the ability to generate keypairs within the Android KeyChain, obtain a certificate from a PKI, and place the issued certificate into the KeyChain. When these calls are used, the private key never exists outside of the Android KeyChain. We submitted our patches to the Android Open Source Project where they are currently under consideration [39].

We extended Google’s Android Basic Managed Profile sample app to demonstrate using our proposed Android Open Source Project enhancements. We embedded Cisco’s open source `libest` library and `estclient` example app binaries into our app. Our app calls our proposed `DevicePolicyManager.generateKeyPair()` method to generate a keypair within the KeyChain, obtains the public key, calls `estclient` to connect to Cisco’s Enrollment over Secure Transport (EST, IETF RFC 7030) test server to obtain a certificate, and calls our proposed `DevicePolicyManager.setCertificate()` method to place the issued certificate into the KeyChain. We are then able to demonstrate the use of the generated keypair and issued certificate with KeyChain-aware apps such as the Chrome web browser and the strongSwan Internet Protocol Security (IPsec) VPN client. We made our example app available as an open source project on GitHub [40].

4 Conclusion and Potential Future Work

Our work demonstrated the ability to improve the security of the Android platform by adding static analysis checks to the tools regularly used by Android app developers in order to help developers identify and correct vulnerabilities up-front in the app development lifecycle, and by building upon the Android's security architecture to reduce or eliminate at the operating system level the potential impact of common app vulnerabilities and malicious behaviors.

We recommend continued efforts to contribute security-related static analysis checks to the Android Open Source Project's lint tool used by Android Studio and the Android Software Development Kit. Additionally, we recommend providing tools within the development environment to help and encourage app developers to leverage emerging Android platform security features such as the Network Security Configuration feature newly extended in the Android N Developer Preview, which allows app developers to declaratively request certificate pinning and other network security features from the platform without the need to make error-prone security customizations within app source code itself.

We also recommend continued efforts to work with the Android Open Source Project to strengthen the security properties enforced upon apps and reduce the impact of app vulnerabilities and malicious behaviors, whether as default behavior or through interfaces that enable app developers to declaratively state each app's security properties. As these improvements are made, we recommend streamlining enterprise app vetting processes to take into account the security protections provided by the device platform.

5 References

- [1] (Viewed 2 July 2015). *Gartner Says Tablet Sales Continue to Be Slow in 2015*. Gartner. Available: <http://www.gartner.com/newsroom/id/2954317>
- [2] (Viewed 4 November 2015). *Security Tips*. Google. Available: <http://developer.android.com/training/articles/security-tips.html>
- [3] (Viewed 4 November 2015). *Improving Your Code with Lint*. Google. Available: <http://developer.android.com/tools/debugging/improving-w-lint.html>
- [4] Fahl et al. (October 2012). *Why Eve and Mallory Love Android: An Analysis of Android SSL (In)Security*. ACM CCS '12.
- [5] Sounthiraraj et al. (February 2014). *Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps*. NDSS '14.
- [6] (August 2014). *SSL Vulnerabilities: Who listens when Android applications talk?* FireEye. Available: <https://www.fireeye.com/blog/threat-research/2014/08/ssl-vulnerabilities-who-listens-when-android-applications-talk.html>
- [7] Montelibano and Dormann. *How We Discovered Thousands of Vulnerable Android Apps in One Day*. RSA Conference USA 2015. Available: https://www.rsaconference.com/writable/presentations/file_upload/hta-t08-how-we-discovered-thousands-of-vulnerable-android-apps-in-1-day_final.pdf
- [8] M. Grace et al. *Unsafe Exposure Analysis of Mobile In-App Advertisements*. WiSec '12. Available: http://www4.ncsu.edu/~mcgrace/WISEC12_ADRISK.pdf
- [9] (Viewed 6 July 2015). *Remote Code Execution as System User on Samsung Phones*. NowSecure. Available: <https://www.nowsecure.com/blog/2015/06/16/remote-code-execution-as-system-user-on-samsung-phones/>
- [10] M. Peck. (4 October 2015). *Add lint check for insecure X509TrustManager implementations*. Android Open Source Project. Available: <https://android.googlesource.com/platform/tools/base/+/c660c577ba072bd94dc20c92f28f9d76102530e1>
- [11] M. Peck. (26 October 2015). *Lint checks for insecure use of SSLCertificateSocketFactory*. Android Open Source Project. <https://android.googlesource.com/platform/tools/base/+/b98fb2952479587aa38b4e260feb2bd454ece3b4>
- [12] M. Peck. (8 October 2015). *Lint check for declaration or use of insecure HostnameVerifier*. Android Open Source Project. <https://android.googlesource.com/platform/tools/base/+/b669f60e6aded9fb50b9025e56e3a007cc482ad7>
- [13] J. Case. (14 April 2011). *Exclusive: Vulnerability in Skype for Android Is Exposing Your Name, Phone Number, Chat Logs, And a Lot More*. AndroidPolice. Available: <http://www.androidpolice.com/2011/04/14/exclusive-vulnerability-in-skype-for-android-is-exposing-your-name-phone-number-chat-logs-and-a-lot-more/>
- [14] J. Van Dyke. (10 August 2015). *World Writable Code Is Bad, MMMMKAY*. NowSecure. Available: <https://www.nowsecure.com/blog/2015/08/10/world-writable-code-is-bad-mmmmkay/>
- [15] M. Peck. (5 October 2015). *Check for calls to getDir with insecure file permissions*. Android Open Source Project. Available:

- <https://android.googlesource.com/platform/tools/base/+3adca6769e92362751721c18c649e9e9c94fbb63>
- [16] M. Peck. (23 October 2015). *Lint checks to identify setting files world-readable or world-writable*. Android Open Source Project. Available: <https://android.googlesource.com/platform/tools/base/+87f30697958a693a063169738aa8fb6c369b8a13>
- [17] M. Peck. (4 October 2015). *Add lint checks for insecure broadcast receivers*. Android Open Source Project. Available: <https://android.googlesource.com/platform/tools/base/+50bfc95d9945178dc8d3e0112e26c8286d47a1ae>
- [18] E. Chin et al. *Analyzing Inter-Application Communication in Android*. MobiSys '11. Available: <https://www.eecs.berkeley.edu/~daw/papers/intents-mobisys11.pdf>
- [19] (6 October 2013). *Secured Broadcasts and SMS Clients*. The CommonsBlog. Available: <https://commonsware.com/blog/2013/10/06/secured-broadcasts-sms-clients.html>
- [20] A. Mahajan. (26 May 2015). *Declare SMS broadcasts as protected to ensure only system apps send those*. Android Open Source Project. Available: <https://android.googlesource.com/platform/packages/services/Telephony/+dcec1e1471a1d32918b5a2beb239693708c548e7>
- [21] (Accessed 9 November 2015). *F-Droid*. Available: <https://f-droid.org/>
- [22] (Accessed 9 November 2015). *Android:usesCleartextTraffic*. Android Open Source Project. Available: <http://developer.android.com/guide/topics/manifest/application-element.html#usesCleartextTraffic>
- [23] J. Kozyrakis. (17 June 2015). *Android M and the war on cleartext traffic*. Available: <https://koz.io/android-m-and-the-war-on-cleartext-traffic/>
- [24] C. Brubaker. (23 October 2015). *Add initial network security config implementation*. Android Open Source Project. Available: <https://android-review.googlesource.com/#/c/179902/>
- [25] V. Tendulkar and W. Enck. (May 2014). *An Application Package Configuration Approach to Mitigating Android SSL Vulnerabilities*. MoST 2014.
- [26] M. Peck. (September 2015). *Add ability to declare certificate pins in application manifest*. Android Open Source Project. Available: <https://android-review.googlesource.com/#/q/Ie357b3e8fd0b5e0f21e1ef9226f94ce945b1cfb8>
- [27] M. Peck. (October 2015). *Add ability to prevent applications from overriding TrustManager*. Android Open Source Project. Available: <https://android-review.googlesource.com/#/q/If6ab0ab309488926cdde7fd55539bd059eea964d>
- [28] M. Peck. (October 2015). *Add ability to prevent applications from overriding HostnameVerifier*. Android Open Source Project. Available: <https://android-review.googlesource.com/#/c/173951/>
- [29] G. Kini. (September 2015). *Add app-level isolatedApplicationData manifest attribute*. Android Open Source Project. Available: <https://android-review.googlesource.com/#/q/I5b9f4f3743f3fa544d14b44ac04549cf87a7ebfe>

- [30] J. Oberheide. (June 2010). *Android Hax*. Summercon 10. Available: <https://jon.oberheide.org/files/summercon10-androidhax-jonoberheide.pdf>
- [31] Poeplau et al. (February 2014). *Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications*. NDSS '14.
- [32] *Alleged Hacking Team Android source code*. (Accessed 9 November 2015). Available: <https://github.com/hackedteam/core-android-market> and <https://github.com/hackedteam/core-android/tree/master/RCSAndroid/jni>
- [33] Maier et al. (September 2014). *Divide-and-Conquer: Why Android Malware cannot be stopped*. ARES 2014. Available: <https://www1.cs.fau.de/filepool/projects/android/divide-and-conquer.pdf>
- [34] G. Kini. (September 2015). <https://android-review.googlesource.com/#/q/Iacc7a5fd7518fcb92d4b029ed4c97c23950038fb>
- [35] (25 August 2015). *Automated PaX exceptions for WebView JavaScript*. Copperhead Security. Available: https://github.com/copperhead/android_frameworks_base/commit/84a0f4e9e312eb0d38e7475853804f99bce35fb5
- [36] (November/December 2013). *Why does Chromium WebView on Android run in single-process mode?* Available: <https://groups.google.com/a/chromium.org/forum/#!topic/chromium-dev/714l6LZyFxx>
- [37] (Accessed 9 November 2015). *KeyChain*. Google. Available: <http://developer.android.com/reference/android/security/KeyChain.html>
- [38] (Accessed 9 November 2015). *Android KeyStore System*. Google. Available: <http://developer.android.com/training/articles/keystore.html>
- [39] M. Peck. (October 2015). *Add ability to generate keypairs within KeyChain*. Android Open Source Project. Available: <https://android-review.googlesource.com/#/q/If41721a58b0d2fd282d194713b828d78a81217ed>
- [40] M. Peck. (October 2015). *Demonstrate generating keypairs within the Android KeyChain*. The MITRE Corporation. Available: <https://github.com/mpeck12/android-BasicManagedProfile/tree/keychain>
- [41] M. Peck. (October 2015). *Demonstrate generating keypairs within the Android KeyChain*. The MITRE Corporation. Available: <https://github.com/mpeck12/android-BasicManagedProfile/tree/est>
- [42] (25 September 2015). *Google Play Warning: SSL Error Handler Vulnerability*. GitHub. Available: <https://github.com/liveservices/LiveSDK-for-Android/issues/63>
- [43] (17 July 2015). *GmsCore_OpenSSL can provoke app rejected from Google Play with all updated?* StackOverflow. Available: <http://stackoverflow.com/questions/31378617/gmscore-openssl-can-provoke-app-rejected-from-google-play-with-all-updated>
- [44] (28 August 2015). *Google Play reminder of deadline for resolving Apache Cordova vulnerabilities*. StackOverflow. Available: <http://stackoverflow.com/questions/32273656/google-play-reminder-of-deadline-for-resolving-apache-cordova-vulnerabilities>

- [45] M. Peck. (November 2015). *Use SELinux to perform authorization checks for privileged KeyChain operations*. Android Open Source Project. Available: <https://android-review.googlesource.com/#/q/I70f3324857cf20ef01d718610d766e8395982ffd>
- [46] (Accessed 9 November 2015). *Android Security Acknowledgements*. Android Open Source Project. Available: <https://source.android.com/devices/tech/security/overview/acknowledgements.html>
- [47] F. Chung. (July 2011). *Custom Class Loading in Dalvik*. Android Developers Blog. Available: <http://android-developers.blogspot.com/2011/07/custom-class-loading-in-dalvik.html>
- [48] (Accessed 9 November 2015). *Sample: hello-jni*. Android NDK. Available: http://developer.android.com/ndk/samples/sample_hellojni.html
- [49] A. Pyles and M. Peck. (November 2015). *Code Execution Demonstration App*. The MITRE Corporation. Available: <https://github.com/mpeck12/custom-class-loader>
- [50] M. Peck. (November 2015). *Lint check to identify calls to Runtime.load() and System.load()*. Android Open Source Project. Available: <https://android-review.googlesource.com/#/c/179980/>
- [51] (November 2015). *Copperhead Security*. Available: <https://twitter.com/CopperheadSec/status/663443145095442432>
- [52] M. Franchin. (22 February 2015). *HKG15-300: Art's Quick Compiler: An unofficial overview*. Linaro. Available: <https://www.youtube.com/watch?v=iho-e7EPHk0>
- [53] L. Armasu. (4 May 2015). *Android Runtime To See Major Performance Boost Thanks to Redesigned Compiler*. Tom's Hardware. Available: <http://www.tomshardware.com/news/android-runtime-art-optimizing-compiler,29035.html>
- [54] N. Hajdarbegovic. (May 2015). *Brace Yourselves Android Developers, A New Android Compiler Is Coming*. Toptal. Available: <http://www.toptal.com/android/brace-yourselves-new-android-compiler-is-coming>
- [55] (Accessed 9 November 2015). *Configuring ART*. Android Open Source Project. Available: <https://source.android.com/devices/tech/dalvik/configure.html>
- [56] M. Chartier. (17 February 2015). *Add JIT*. Android Open Source Project. Available: <https://android-review.googlesource.com/#/c/123156/>
- [57] (Accessed 12 March 2016). *Network Security Configuration*. Google. Available: <http://developer.android.com/preview/features/security-config.html>
- [58] J. Kozyrakis. (15 February 2016). *Network Security Policy configuration for Android apps*. Available: <https://koz.io/network-security-policy-configuration-for-android-apps/>
- [59] A. Ludwig. (March 2016). *Building an Android Scale Incident Response Process*. RSA Conference 2016. Available: https://www.rsaconference.com/writable/presentations/file_upload/mbs-r03-building-an-android-scale.pdf
- [60] (Accessed 12 March 2016). *Android targetSdkVersion documentation*. Google. Available: <http://developer.android.com/guide/topics/manifest/uses-sdk-element.html#target>

- [61] (Accessed 12 March 2016). *Android provider element documentation*. Google. Available: <http://developer.android.com/guide/topics/manifest/provider-element.html>
- [62] M. Peck. (Accessed 12 March 2016). *Add minTargetSdkVersion input selector*. Android Open Source Project. Available: <https://android-review.googlesource.com/#/q/Ib9f6ded9bd2f426861a6d843861b4074084253b0>
- [63] M. Peck. (Accessed 12 March 2016). *Add untrusted app legacy domain, make untrusted app stricter*. Android Open Source Project. Available: <https://android-review.googlesource.com/#/c/195590/>
- [64] (Accessed 14 March 2016). *Profile-guided JIT/AOT compilation*. Google. Available: http://developer.android.com/preview/api-overview.html#jit_aot

Appendix A Using the New Lint Checks

All of our proposed lint checks have now been merged into the Android Open Source Project and included in the current beta releases of Android Studio 2.0 and the Android Plugin for Gradle. This appendix provides historical information of how to separately compile the lint checks and include them in the Android development environment as a jar plugin through the following steps:

- Download the source code for the desired lint checks from the Android Open Source Project: <https://android.googlesource.com/platform/tools/base/+log/studio-master-dev/lint/libs/lint-checks/src/main/java/com/android/tools/lint/checks>
- Place the source code in its own directory tree, e.g. in a directory called “androidlint”
- Change the package names in the source code to reflect the created directory tree (e.g. change the package entry at the top of each source code file to a value such as “package androidlint;”)
- Create a `MyIssueRegistry.java` with contents similar to the below, with an entry in the array for each Issue declared in the lint check source code:

```
package androidlint;

import java.util.List;
import java.util.Arrays;

import com.android.tools.lint.client.api.IssueRegistry;
import com.android.tools.lint.detector.api.Issue;

public class MyIssueRegistry extends IssueRegistry {
    @Override
    public List<Issue> getIssues() {
        return Arrays.asList(
            TrustAllX509TrustManagerDetector.ISSUE,
            UnsafeBroadcastReceiverDetector.ACTION_STRING);
    }
}
```

- Compile the lint checks and issue registry, e.g. `javac -cp <sdk-path>/tools/lib/lint-api.jar *.java` where <sdk-path> is the installed location of the Android SDK
- Create a `MANIFEST.MF` file with contents similar to the below:

```
Manifest-Version: 1.0
Lint-Registry: androidlint.MyIssueRegistry
```

- Bundle the compiled lint checks, issue registry, and MANIFEST.MF file into a jar file by running: `jar cmf MANIFEST.MF custom.jar androidlint/*.class`
- Create an `.android/lint` directory under the user's home directory and copy the jar file to it, e.g. `cp custom.jar /home/<username>/.android/lint/custom.jar` on most Linux distributions, or `copy custom.jar C:\Users\<username>\.android\.lint\custom.jar` on Windows.
- Run `lint -list Security` and verify that the new lint checks appear in the list. They will now be used by default when running lint from the command line (e.g. with `lint` or `gradlew lint`). Unfortunately, additional steps are needed to integrate the lint checks directly into the Android Studio UI.

Appendix B Demonstration Application

We wrote an application based on Google’s sample custom class loading app [47] and the hello-jni sample app found in the Android Native Development Kit (NDK) [48] that demonstrates the ability to download and execute Dalvik bytecode and native code from arbitrary websites. The app demonstrates several of the security vulnerabilities that are identified by our Android lint checks and/or are mitigated by our proposed Android OS security enhancements.

The app deliberately performs several poor security practices:

- Use of plaintext http rather than https to download code, enabling susceptibility to man-in-the-middle attacks
- Toggle-able ability to use an insecure `X509TrustManager` that does not validate the server’s X.509 certificate when connecting over https, enabling susceptibility to man-in-the-middle attacks
- Toggle-able ability to store downloaded files as world-readable and world-writable, opening the files up to manipulation by other apps installed on the device

More information (including how to compile and use the app) can be found in our GitHub repository [49].

Appendix C Android Runtime (ART) Additional Discussion

C.1 ART Background

In Android 4.4, the Android runtime switched from Dalvik to Android Runtime (ART). Dalvik used a bytecode interpreter along with an always-enabled just-in-time (JIT) compiler. The focus of ART is ahead-of-time (AOT) compilation. With AOT compilation, application bytecode is compiled prior to execution into executable code optimized to the particular device platform. The AOT compilation is accomplished with the `dex2oat` compiler installed on Android devices, where OAT refers to the file format of the optimized native executable code produced by the compiler. AOT compilation occurs when a new application is installed and can also occur when an operating system update is installed. In the case of a system update where all applications may need to be recompiled, AOT compilation can introduce undesired delays at device boot time, particularly if full optimization is enabled.

Among the various `dex2oat` compiler options, we focus on two: `--compiler-backend` and `--compiler-filter`. The `--compiler-backend` option refers to either “quick” or “optimized”. The default option is “quick”. The “optimized” compiler appeared to still be under development and not available for use as of Android versions 5.0-5.1 [52] [53] [54].

The “quick” compiler has two levels of Intermediate Representations (IR) [52]. The compilation flow goes from dex code to Mid Level IR (MIR), to IR and finally to OAT format. This multi-stage process can have different behavior depending upon the optimization levels (compilation time and disk space are factors) selected. Some optimization levels can potentially result in some methods omitted from the final OAT output.

Additionally, the “quick” compiler has several optimizations that can be fed in through the `--compiler-filter` options. The options range from no compilation to full compilation that compiles all methods. The “interpret-only” option skips all compilation and relies solely on the interpreter, which is probably the fastest AOT option with respect to installation time. The “everything” option compiles almost all methods including rare methods. The full range of options is described in [55].

The AOT compiler’s behavior in ART ranges between a long wait time at app installation and system update time and larger size on disk (“everything”) to quick installation with minimal compilation options (“interpret-only”), which largely mimics the older Dalvik behavior. The default option in Android version 5.0 is the “speed” option. This method is designed to “compile most methods and maximize runtime performance” [55].

C.2 JIT Compilation in ART

We observed from the Android Open Source Project source code commit history that just-in-time (JIT) compilation was introduced into ART in February 2015 [56]. JIT compilation is controlled by the `dalvik.vm.usejit` system property, and is currently only enabled by default in engineering builds (these are special builds not used in production device deployments). The motivation behind the inclusion of a JIT compiler as well as security concerns are addressed in the following paragraphs.

C.2.1 Motivation

The `--compiler-filter` options within the ART AOT compiler provide a range of behaviors that the ART runtime has to be able to handle. For instance, the runtime behavior for

the “interpreter-only” option will perform a lot slower than the runtime behavior for the “everything” option. We speculate that the JIT compiler was added to ART to handle these cases where major portions (or all) of the application code were not included in the ahead-of-time compilation. The worst-case scenario for runtime performance presumably would be to run ART with the “interpreter-only” flag, which without JIT compilation would require all code to be interpreted at runtime.

It seems logical that JIT may also improve the behavior of the default “speed” compilation flag. As described earlier, most methods are compiled, but it is possible that certain “hot” methods which are missed with the AOT compilation phase are called regularly and would benefit from JIT compilation.

C.2.2 Security Risks

Reintroducing JIT into the ART runtime poses some security risks. Use of JIT adds memory mappings with RWX security permissions to every application. It may be possible for an attacker (running within the same application) to locate the RWX memory region and add arbitrary code.

We wrote proof-of-concept code that runs as a native library within an Android application. This code (which doesn’t need any special permissions) reads from `/proc/self/maps` and locates the map with RWX permissions. It is then straightforward to locate the JIT mapping and alter it arbitrarily.

Additionally, as previously discussed in Section 3.3.1, JIT compilation interferes with the ability to impose executable memory security protections upon applications.

Appendix D Memory Mapping Examples

The below sections provide examples of memory mappings observed in Android apps. These examples are meant to supplement the discussion in Section 3.3.1.

D.1 Memory Mappings of Chrome App

Below in Figure 5, we show the results of examining the memory mappings of the Chrome web browser Android app. The results indicate that the writable and executable mappings are only present in the processes running in the `isolated_app` security context.

```
# ps -Z |grep chrome
u:r:untrusted_app:s0          u0_a94      2195   263    com.android.chrome
u:r:isolated_app:s0          u0_i38      5911   263    com.android.chrome:
sandboxed_process1
u:r:isolated_app:s0          u0_i40      6431   263    com.android.chrome:
sandboxed_process2
u:r:untrusted_app:s0          u0_a94      25892
263    com.android.chrome:privileged_process2
# grep rwx /proc/2195/maps
# grep rwx /proc/5911/maps
2700a000-270ff000 rwxp 00000000 00:00 0
2ba0a000-2ba0b000 rwxp 00000000 00:00 0
2c30a000-2c3ff000 rwxp 00000000 00:00 0
2f60a000-2f648000 rwxp 00000000 00:00 0
37a0a000-37a0b000 rwxp 00000000 00:00 0
37f0a000-37fff000 rwxp 00000000 00:00 0
7ab0a000-7ab0b000 rwxp 00000000 00:00 0
7e20a000-7e2ff000 rwxp 00000000 00:00 0
7ee0a000-7eeff000 rwxp 00000000 00:00 0
7f20a000-7f2ff000 rwxp 00000000 00:00 0
7f90a000-7f9ff000 rwxp 00000000 00:00 0
8140a000-814ff000 rwxp 00000000 00:00 0
# grep rwx /proc/6431/maps
2b10a000-2b10b000 rwxp 00000000 00:00 0
38e0a000-38eff000 rwxp 00000000 00:00 0
3b10a000-3b10b000 rwxp 00000000 00:00 0
3e90a000-3e948000 rwxp 00000000 00:00 0
7a90a000-7a90b000 rwxp 00000000 00:00 0
7ea0a000-7eaff000 rwxp 00000000 00:00 0
80a0a000-80aff000 rwxp 00000000 00:00 0
# grep rwx /proc/25892/maps
#
```

Figure 5: Memory Mappings of Chrome App

D.2 Memory Mappings of Application Using WebView

Below in Figure 6, we show the results of examining the memory regions of the com.audible.application Android app. It uses Android WebView with JavaScript enabled, causing the app to map memory regions as both writable and executable:

```
# ps -Z |grep audible
u:r:untrusted_app:s0          u0_a233    18607
263    com.audible.application

# grep rwx /proc/18607/maps
3810a000-3810b000 rwxp 00000000 00:00 0
38f0a000-38f0b000 rwxp 00000000 00:00 0
8230a000-8230b000 rwxp 00000000 00:00 0
8260a000-8266a000 rwxp 00000000 00:00 0
8560a000-856ff000 rwxp 00000000 00:00 0
```

Figure 6: Memory Mappings of Audible App